Smart Pointers Workshop Solutions

Copyright © James Raynard 2021

Shared Pointer Implementation

- Implement a simplified version of std::shared_ptr, without some of the more elaborate features:
  - The control block is a single int, which is the reference counter
  - Not threadsafe
  - Only the * and -> operators are implemented
  - No equivalent to make_shared()
  - No move operators
- Write a simple program to test your class

Copyright © James Raynard 2021

Class Definition

- We will need a pointer to the allocated memory
- We will also need a pointer to int, to simulate the control block

```
template <typename T>
class shared_pointer {
        T *data;                    // The allocated memory
        int *counter;               // The reference counter
public:
   shared_pointer(T *t);
   ~shared_pointer();
   shared_pointer(const shared_pointer& other);
   shared_pointer& operator=(const shared_pointer& other);
   T& operator *() const;
   T operator ->() const;
};
```

Copyright © James Raynard 2021

Shared Pointer Implementation

- Implement a simplified version of std::shared_ptr, without some of the more elaborate features:
  - The control block is a single int, which is the reference counter
  - Not threadsafe
  - Only the * and -> operators are implemented
  - No equivalent to make_shared()
  - No move operators
- Write a simple program to test your class

Copyright © James Raynard 2021

Constructor

- The constructor takes a pointer argument, representing the allocated memory
- The constructor will adopt this pointer argument
- It will also allocate and initialize the reference counter
- This is the first object which uses the allocated memory, so the initial value of the counter will be 1

```
shared_pointer(T *t) : data(t), counter(new int(1)) {}
```

Copyright © James Raynard 2021

Destructor

- The destructor must decrement the reference counter, since there is now one less object using the shared memory
- If this makes the reference counter equal to 0, then this is the last object using the shared memory
- In that case, the shared memory and the reference counter must be released

```
~shared_pointer() {
   if (--*counter == 0) {
      delete data;
      delete counter;
   }
}
```

Copyright © James Raynard 2021

Copy Constructor

- The copy constructor creates another object which uses the shared memory
- It must therefore increment the reference counter
- It also initializes the new object to use the same data and counter as its argument

```
shared_pointer(const shared_pointer& other) : data(other.data), counter(other.counter) {
   ++*counter;
}
```

Copyright © James Raynard 2021

Assignment Operator

- If the two objects already share the same allocated memory, nothing needs to be done
  - The assigned-to object already has the correct pointer values
  - The number of objects using the allocated memory does not change, so the reference counter must not be modified

Copyright © James Raynard 2021

Assignment Operator

- If the two objects do not share the same allocated memory:
  - The assigned-to object must have its reference counter decremented
  - If this makes the reference counter equal to 0, then its pointers have to be deleted
  - Its pointers are then reassigned so that it shares the allocated memory and reference counter of the assigned-from object
  - The shared reference counter must be incremented, since there is now one more object which accesses the allocated memory

Copyright © James Raynard 2021

Dereferencing Operators

- The dererefencing operators simply refer the data in the allocated memory

```
T& operator *() const { return *data; }
T operator ->() const { return *data; }
```

Copyright © James Raynard 2021